


Differential Verification of Information Flow in SEAndroid Policies

Lorenzo Ceragioli(✉), Letterio Galletta, and Edoardo Lunati

IMT School for Advanced Studies Lucca, Lucca, Italy
{lorenzo.ceragioli,letterio.galletta,edoardo.lunati}@imtlucca.it
lorenzo.ceragioli@imtlucca.it, letterio.galletta@imtlucca.it,
edoardo.lunati@imtlucca.it

Abstract. SEAndroid is deployed on almost all modern Android devices to enforce mandatory access control. The Android Open Source Project regularly publishes a predefined SEAndroid policy that each vendor customises for its devices. These customisations and policy updates may cause security regressions: even small misconfigurations may introduce new information flows leading to exploitable vulnerabilities. We propose a differential verification framework that determines whether security-relevant changes between two SEAndroid configurations affect information-flow properties as intended. Our verification framework is based on the novel comparative modal logic CML interpreted over pairs of Kripke structures. We formalise SEAndroid policies within this logic and define a suitable model-checking algorithm for differential reasoning. We implement our framework in the tool *Mordente*, and evaluate it on real-world SEAndroid policies.

Keywords: Comparative modal logic · Policy evolution analysis · Access control

1 Introduction

Android is currently the most widely used operating system for mobile devices, making it a frequent target of privilege escalation and code execution attacks [7, 16]. Typically, these attacks aim to compromise high-privileged system daemons, often bypassing the traditional discretionary access control of Linux systems [7]. To mitigate such threats, Security Enhancements for Android (SEAndroid), based on SELinux [2], introduces mandatory access control into the Android platform. SEAndroid is integrated into the kernel and checks every action performed by a subject (e.g., a process) over an object (e.g., a file or socket). In SEAndroid, subjects and objects are associated with abstract security labels as specified in a labelling policy. In addition, a permission policy lists the operations that a labelled subject may perform on a labelled object (all others being denied by default). The Android Open Source Project (AOSP) continuously updates a default SEAndroid policy, which the Android Original Equipment Manufacturers

A. Sampaio and M. Stoelinga (Eds.): FM 2026, LNCS 16556, pp. 461–479, 2026.
https://doi.org/10.1007/978-3-032-26204-2_24

(OEMs) further customise for their devices. Given the size and evolving nature of the Android codebase, ensuring that the SEAndroid configuration enforces the intended security properties is a challenging task [11]. Additionally, modifications to the permission and labelling policies affect the system's effective security posture, and both AOSP and OEMs have introduced misconfigurations and vulnerabilities in the past [14].

Previous work has investigated the security of SEAndroid and SELinux policies, proposing tools for visualisation, inspection, and static verification [5,9,10,12,15]. These tools formalise security in terms of information flow, a well-established approach for reasoning about integrity, confidentiality, and isolation. In practice, we say that there is an information flow from file f to file f' if some processes $p_1, p_2, \ldots, p_n$ and files $f_1, f_2, \ldots, f_{n+1}$ exist such that the process p_i reads from f_i and writes on f_{i+1}, with $f_1 = f$ and $f_{n+1} = f'$. Several important security properties for Android systems can be stated in terms of information flow. For example, to avoid privilege escalation, we can impose that low-privileged domains, such as `untrusted_app`, can never write to `system_file`, `exec_file`, or `shared_libs_file`. Moreover, to enforce cross-app isolation, we may prohibit `untrusted_app` or `platform_app` from interacting with `app_data_file` belonging to other applications. However, existing tools provide little support for comparing configurations across versions or vendors in terms of their security properties. Additionally, they do not help analysts preserve security properties when they are undocumented, which is often the case in real-world policies. While one could verify each version independently against a fixed security specification, this approach requires such a specification to be explicitly provided and fully documented. This situation is rarely met in practice, where security properties are often implicit and undocumented. Instead, directly comparing two configurations allows analysts to focus precisely on whether the changes introduced between versions have the intended security impact, without the need to reason about the entire policy from scratch.

To address this gap, we propose a differential verification framework that determines whether security-relevant changes between two SEAndroid configurations alter information-flow properties as intended. Our framework is based on the new *comparative modal logic* CML, interpreted over sets of Kripke structures representing different system versions. CML extends classical logic with three classes of operators: (*i*) two forward modalities $\Diamond$ ("for some next state") and $\Box$ ("for all next states"); (*ii*) two backward modalities $\blacklozenge$ ("for some previous state") and $\blacksquare$ ("for all previous states"); (*iii*) a version operator $\uparrow_i$ used to refer to the i-th version of the system, represented by the i-th Kripke structure. A peculiarity of CML is that, to decide the validity of a formula where two operators $\uparrow_i$ and $\uparrow_j$ occur (with $i \neq j$), the properties stated in terms of two different Kripke structures must be compared. We do this by enriching each of our Kripke structures with an interpretation over a common set Ω of concrete states.

To use CML for differential reasoning over SEAndroid policies, we first compute a relation describing the information flows between file labels permitted by each policy. The Kripke structure of a SEAndroid configuration consists of

```
1 (A/.*)|(.*/b)      a
2 C/a                b
3 B/b                c
4 C/b                d
```

(a) File labeling policy of $\mathcal{C}_1$.

```
1 (A/.*)|(.*/b)      a
2 .*/a               e
3 C/b                d
```

(b) File labeling policy of $\mathcal{C}_2$.

```
1 allow p1 a : file { write };
2 allow p1 b : file { read };
3 allow p2 c : file { read write }
4 allow p2 a : file { setattr };
5 allow q1 b : file { read };
6 allow q1 d : file { write };
7 allow q2 d : file { write };
8 allow q2 c : file { getattr };
```

(c) Access control policy of $\mathcal{C}_1$.

```
1 allow p e : file { read };
2 allow p a : file { append };
3 allow q e : file { read };
4 allow q d : file { write };
```

(d) Access control policy of $\mathcal{C}_2$.

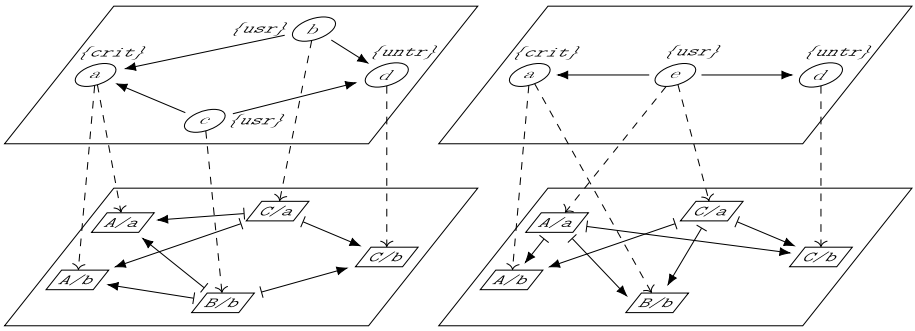
Fig. 1. Two examples of SEAndroid configurations.

file labels, the corresponding information-flow relation, and a state annotation capturing relevant security properties. In the context of SEAndroid, the set Ω of concrete states contains all the file paths, and the interpretation of Kripke structures derives from the labelling policy: file labels are assigned to file paths by matching regular expressions. This allows us to relate abstract labels to actual filesystem objects and reason about information flows in the system. Finally, we implement our framework in the tool *Mordente*, available at [1], and evaluate it on real-world SEAndroid policies of different vendors and versions.

To the best of our knowledge, our work is the first proposal comparing different versions of SEAndroid policies in terms of their information flow, taking into account the changes in the two labelling policies. A key advantage of our approach over other tools for regression verification is that it does not require the policy analyst to provide a complete specification of the desired security properties. When changing from one SEAndroid version to another, *Mordente* can verify that the update: (*i*) impacts the modified portion of the system as desired; (*ii*) preserves the needed permissions (these flows can only increase); (*iii*) does not add dangerous permissions (these flows can only decrease). Note that (*i*) allows expressing novel desired properties by stating them in terms of the problems of the previous version, whereas (*ii*) and (*iii*) encode the preservation of functional and security properties, including those not explicitly documented by the developers. Combining the last two points, one can verify that the update only impacts a portion of the system, leaving the remaining part unaltered.

***Mordente* at Work.** Consider the two SEAndroid configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ of Fig. 1. Each configuration consists of two components: a *labelling policy* (above in the figure) and a *permission policy* (below in the figure).

The labelling policy is stored in a so-called *file-context* and consists of a sequence of rules mapping regular expressions of filesystem paths to the security context that labels them. Security contexts usually consist of a user, a role,



(a) File information flow in $\mathcal{C}_1$. (b) File information flow in $\mathcal{C}_2$.

Fig. 2. Kripke structures of two SEAndroid configurations: solid arrows represent information flows, dashed arrows represent file labelling.

and a type, but for simplicity, we consider only the type component, which is the most relevant in Android (being type enforcement the primary access control mechanism in mobile devices); thus, hereafter, we treat labels as consisting solely of types. For example, in Fig. 1a the rule on line 1 assigns label `a` to all files in the directory `A` and to any file named `b`, while line 3 assigns `c` to just the file at path `B/b`. Note that the order of rules matters, and SEAndroid applies the last-matched one (regexes matching smaller languages are usually at the end of file contexts). Thus, `B/b` is associated with `c` (line 3) and not with `a` (line 1).

The permission policy specifies which operations labelled subjects are allowed to perform on labelled objects. Administrators typically specify configurations using SELinux's kernel policy language [4], which is then compiled to a kind of (kernel binary) access-control matrix. As illustrated in Fig. 1a, the permission policy is essentially a sequence of rules of the following form.

```
allow subject_type object_type : class {permissions}
```

For instance, the rule on line 1 permits subjects labelled `p1` writing files labeled `a`. Any operation not explicitly allowed is denied by default. Permission policies may also contain `neverallow` rules, attributes that group related types under shared behavioural constraints, and macros that expand into commonly used sets of permissions. The full policy language is outside the scope of this paper, as it is used only for examples, since our tool operates on compiled policies.

In order to compare the two configurations of Fig. 1, we first give them a semantics in terms of Kripke structures (see Fig. 2) where states represent file labels, and arcs the information flow between them induced by *read* and *write* operations (or similar ones like *getattr* and *setattr*). Then, we encode security properties using three atomic propositions: `crit` (i.e. *critical*, is associated with `a` both in $\mathcal{C}_1$ and $\mathcal{C}_2$), `untr` (i.e. *untrusted*, is associated with `d` both in $\mathcal{C}_1$ and $\mathcal{C}_2$) and `usr` (i.e. *user*, is associated with all the remaining labels). Note that label names differ in the two configurations, and the common ones may have a

different meaning (like `a`). Therefore, in order to compare the semantics of the two configurations, their labels must be concretised in terms of actual file paths. This is achieved through the file labelling policies: for each policy, we evaluate the labels while preserving the edges, obtaining the graphs below in Fig. 2.

Using *Mordente*, the policy administrator can build these transition systems automatically, and can check the validity of the following CML proposition ϕ, stating that all the files that used to affect some untrusted file in $\mathcal{C}_1$ satisfy `usr` in $\mathcal{C}_2$, and they still influence the same files of the previous version.

$$\phi = \uparrow_1(\Diamond \texttt{untr}) \Rightarrow \uparrow_2(\texttt{usr} \wedge \Diamond \uparrow_1 \texttt{untr})$$

The tool signals that ϕ is not satisfied by the version update, informing that the files labelled as `c` in $\mathcal{C}_1$ and as `a` in $\mathcal{C}_2$ contradict the property, e.g. the concrete file path `B/b`. Note indeed that in Fig. 2a, the file at path `B/b` accesses `C/b`, which is untrusted, while the same access is not possible in Fig. 2b.

Synopsis. In Sect. 2, we define CML and its model checking. Section 3 encodes SEAndroid using our logic. Section 4 presents *Mordente* and its experimental evaluation. In Sects. 5 and 6, we compare our approach with the literature and draw some conclusions.

2 A Comparative Modal Logic

We define Comparative Modal Logic (CML), a logic for comparing different versions of the same system, each represented as a transition system with states labelled by the properties of interest (a Kripke structure). In the following, we let $\mathbb{I}$ be the finite set of version names, and $\mathbb{P}$ the set of atomic propositions.

Definition 1 (CML Proposition). *A* CML *proposition $\phi \in \Phi$ is defined by the following grammar, with $i \in \mathbb{I}$ a version name and $p \in \mathbb{P}$ an atomic proposition.*

$$\phi :: \mathit{true} \mid p \mid \phi \wedge \phi \mid \neg\phi \mid \uparrow_i \phi \mid \Diamond\phi \mid \blacklozenge\phi$$

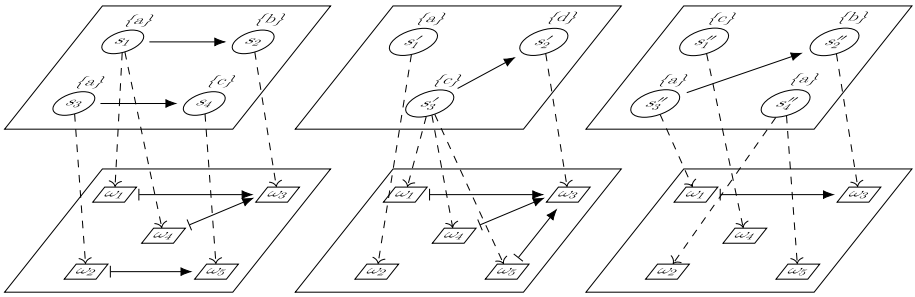
Intuitively: *true* is always satisfied; p tells that the atomic proposition p is satisfied; $\uparrow_i \phi$ is satisfied if ϕ is satisfied in the version i; $\Diamond\phi$ is satisfied if ϕ is satisfied in at least a next state; $\blacklozenge\phi$ is satisfied if ϕ is satisfied in at least a previous state; boolean conjunction and negation behaves as expected.

We derive other logical operators in the usual way.

$$\Box\phi = \neg\Diamond\neg\phi \qquad \blacksquare\phi = \neg\blacklozenge\neg\phi \qquad \phi \vee \psi = \neg(\neg\phi \wedge \neg\psi) \qquad \phi \Rightarrow \psi = \neg\phi \vee \psi$$

Example 1. Intuitively, the formula $\uparrow_1(\texttt{a} \wedge \Diamond(\neg\texttt{b} \wedge \uparrow_2 \texttt{b}))$ is satisfied by a pair of versions of the same system if, in the first version, it is true that a is valid and we can access some state where b is valid only in the second version.

To give the CML semantics, we first define a model for versions in isolation.



(a) Version 1. (b) Version 2. (c) Version 3.

Fig. 3. A CML model of example and the concretization its components.

Definition 2 (Kripke Structure). *A* Kripke structure $\mathcal{K} = (S, \rightarrow, \vdash)$ *is a triple with:* S *a set of states;* $\rightarrow \subseteq S \times S$ *the transition relation; and where* $\vdash \subseteq S \times \mathbb{P}$ *associates atomic propositions with states in which they hold.*

The purpose of CML is to compare different versions, and for that we need a common ground, which is represented by the special set of states $\Omega \ni \omega, \omega', \ldots$ that we assume to be *concrete states*. The states of each version can be thought of as abstract representations of classes of concrete states.

Definition 3 (State Evaluation). *A* state evaluation *for a set of states* S *is a function* $\downarrow\colon S \longrightarrow 2^{\Omega}$ *that associates each state with the concrete ones it represents, and it satisfies that* $\downarrow(s) \cap \downarrow(s') = \emptyset$ *if* $s \neq s'$*, and* $\bigcup_{s \in S} \downarrow(s) = \Omega$*.*

As a result of that, the Kripke structure of a version $\mathcal{K}$ can always be seen as an abstract representation of a Kripke structure over the set of concrete states: its concretisation is obtained by evaluating states according to the given function.

Definition 4 (Concretization). *Given a Kripke structure* $\mathcal{K} = (S, \rightarrow, \vdash)$ *and a state evaluation* $\downarrow$ *for* S*, the concretization of* $\mathcal{K}$ *with respect to* $\downarrow$ *is a Kripke structure* $(\Omega, \mapsto, \Vdash)$ *with*

- $\mapsto \subseteq \Omega \times \Omega$ *such that* $\omega \mapsto \omega'$ *if and only if* $s \rightarrow s'$ *for some* s *and* s' *in* S *with* $\omega \in \downarrow(s)$ *and* $\omega' \in \downarrow(s')$*;*
- $\Vdash \subseteq \Omega \times \mathbb{P}$ *such that* $\omega \Vdash p$ *if and only if* $\omega \in \downarrow(s)$ *and* $s \vdash p$ *for some* $s \in S$*.*

A CML model is a set of Kripke structures paired with state evaluations.

Definition 5 (Model). *A* CML *model* $\mathcal{M} = \{(\mathcal{K}_i, \downarrow_i)\}_{i \in \mathbb{I}}$ *is a collection of pairs indexed by the set* $\mathbb{I}$ *of versions, where* $\mathcal{K}_i = (S_i, \rightarrow_i, \vdash_i)$ *is the Kripke structure of the version* i*, and* $\downarrow_i\colon S_i \longrightarrow 2^{\Omega}$ *is its state evaluation.*

Example 2. Let $\Omega = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5\}$. The CML model in Fig. 3 is $\mathcal{M} = \{(\mathcal{K}_1, \downarrow_1), (\mathcal{K}_2, \downarrow_2), (\mathcal{K}_3, \downarrow_3)\}$, where, e.g., the graph above in Fig. 3b represents $\mathcal{K}_2 = (\{s'_1, s'_2, s'_3\}, \{(s'_3, s'_2)\}, \{(s'_1, a), (s'_2, d), (s'_3, c)\})$, while the graph below is its concretisation with respect to the state evaluation represented by the vertical dashed arrows, $\downarrow_2 = \{(s'_1, \{\omega_2\}), (s'_2, \{\omega_3\}), (s'_3, \{\omega_1, \omega_4, \omega_5\})\}$.

We now define the validity of a formula according to a CML model. By expressing it in terms of concrete states, we allow for comparing different versions.

Definition 6. *Given a* CML *model* $\mathcal{M} = \{(\mathcal{K}_i, \downarrow_i)\}_{i \in I}$*, we define the validity of a* CML *proposition* $\phi \in \Phi$ *with respect to a concrete state* $\omega \in \Omega$ *and a version* $i \in I$ *though the following satisfaction relation* $\omega, i \models_{\mathcal{M}} \phi$*:*

$$
\begin{array}{ll}
\omega, i \models_{\mathcal{M}} \mathit{true} & \\
\omega, i \models_{\mathcal{M}} p & \text{if } \omega \Vdash_i p \\
\omega, i \models_{\mathcal{M}} \phi \wedge \phi' & \text{if } \omega, i \models_{\mathcal{M}} \phi \text{ and } \omega, i \models_{\mathcal{M}} \phi' \\
\omega, i \models_{\mathcal{M}} \neg\phi & \text{if } \omega, i \not\models_{\mathcal{M}} \phi \\
\omega, i \models_{\mathcal{M}} \uparrow_j \phi & \text{if } \omega, j \models_{\mathcal{M}} \phi \\
\omega, i \models_{\mathcal{M}} \Diamond\phi & \text{if } \omega' \text{ exists such that } \omega \mapsto_i \omega' \text{ and } \omega', i \models_{\mathcal{M}} \phi \\
\omega, i \models_{\mathcal{M}} \blacklozenge\phi & \text{if } \omega' \text{ exists such that } \omega' \mapsto_i \omega \text{ and } \omega', i \models_{\mathcal{M}} \phi
\end{array}
$$

where $(\Omega, \mapsto_i, \Vdash_i)$ *is the concretisation of* $\mathcal{K}_i$ *with respect to* $\downarrow_i$*.*

We say that a concrete state $\omega \in \Omega$ *satisfies a* CML *proposition* ϕ*, and we write* $\omega \models_{\mathcal{M}} \phi$*, if* $\omega, i \models_{\mathcal{M}} \phi$ *for any* $i \in \mathbb{I}$*. Finally, we say that the* CML *model* $\mathcal{M}$ *satisfies* ϕ*, and we write* $\mathcal{M} \models \phi$*, if* $\omega \models_{\mathcal{M}} \phi$ *for any* $\omega \in \Omega$*.*

Example 3. Consider the CML model $\mathcal{M}$ of Example 2 and assume that we want to preserve some property while passing from version i to j. The property encoded by the atomic predicate a is preserved if $\uparrow_i \mathtt{a} \Rightarrow \uparrow_j \mathtt{a}$, i.e. when all concrete states satisfying $\mathtt{a}$ in the first version also satisfy it in the second one. For example $\mathcal{M} \models \uparrow_2 \mathtt{a} \Rightarrow \uparrow_1 \mathtt{a}$, while $\mathcal{M} \not\models \uparrow_1 \mathtt{a} \Rightarrow \uparrow_3 \mathtt{a}$ (the state ω_4 is a counterexample).

The preservation of reachability is more involved: various CML propositions can be used, with different nuances in their meaning. Consider the following predicates, noting that $\uparrow_i(\mathtt{a} \wedge \Diamond \mathtt{b})$ is satisfied by concrete states that used to satisfy $\mathtt{a}$ and to access states that used to satisfy $\mathtt{b}$ in the old version i:

- $\phi_{i,j} = \uparrow_i(\mathtt{a} \wedge \Diamond \mathtt{b}) \Rightarrow \uparrow_j(\mathtt{a} \wedge \Diamond \mathtt{b})$ specifies that such states still satisfy $\mathtt{a}$ and can access states satisfying $\mathtt{b}$ now;
- $\psi_{i,j} = \uparrow_i(\mathtt{a} \wedge \Diamond \mathtt{b}) \Rightarrow \uparrow_j \Diamond \uparrow_i \mathtt{b}$ means that they replicate the accessibility of the previous version, but says nothing about satisfying atomic predicates;
- $\xi_{i,j} = \uparrow_i(\mathtt{a} \wedge \Diamond \mathtt{b}) \wedge \uparrow_j \mathtt{a} \Rightarrow \uparrow_j \Diamond \mathtt{b}$ requires only states that currently satisfy $\mathtt{a}$ to replicate the accessibility of the previous version.

Then, $\mathcal{M}$ satisfies $\psi_{1,2}$, $\xi_{1,2}$ and $\xi_{1,3}$ (see Fig. 3), but not $\phi_{1,2}$ nor $\phi_{1,3}$ (ω_4 does not satisfy a in $\mathcal{K}_2$ nor $\mathcal{K}_3$), and not $\psi_{1,3}$ (ω_4 does not access ω_3 in $\mathcal{K}_3$).

We now investigate the model-checking problem, namely, given a model $\mathcal{M}$ and a CML proposition ϕ, we want to check if $\omega \models_{\mathcal{M}} \phi$ for any $\omega \in \Omega$. Ideally, we want to compute the satisfaction set $\{\omega \in \Omega \mid \omega \models_{\mathcal{M}} \phi\}$, but Ω is often considerably large (even infinite in theory): we need a feasible way of computing a succinct representation of the satisfaction set in terms of the (finite and possibly small) sets of states S_i.

We start by defining a normal form for CML where $\uparrow_i$ appears only before modal operators and atomic propositions, and before each of them. Formally:

Definition 7 (Normal Form). CML *propositions in* normal form *are generated by the following grammar, with $i \in \mathbb{I}$ a version and $p \in \mathbb{P}$ atomic proposition.*

$$\phi :: true \mid \phi \wedge \phi \mid \neg\phi \mid \uparrow_i p \mid \uparrow_i \Diamond\phi \mid \uparrow_i \blacklozenge\phi$$

We let $\Phi^\bullet \subsetneq \Phi$ be the set of CML *propositions in normal form.*

Each CML proposition can be transformed into an equivalent normal form.

Theorem 1. *For each* CML *proposition ϕ, there exists a* CML *proposition in normal form $\phi^\bullet$ such that $\omega \models_{\mathcal{M}} \phi$ if and only if $\omega \models_{\mathcal{M}} \phi^\bullet$ for any ω and $\mathcal{M}$.*

We define compatible states, i.e. functions associating a state to each version i such that it is possible to be in all of the states at the same time. We write $\bar{s}$ for a function from versions to states, and $\bar{s}[i]$ for the state associated with $i \in \mathbb{I}$.

Definition 8 (Compatible State). *Given a* CML *model $\mathcal{M} = \{(\mathcal{K}_i, \downarrow_i)\}_{i\in\mathbb{I}}$ with $\mathcal{K}_i = (S_i, \rightarrow_i, \vdash_i)$ for all $i \in \mathbb{I}$, a* compatible state *of $\mathcal{M}$ is a function $\bar{s}$ associating a version $i \in \mathbb{I}$ with a state $s \in S_i$ such that $\bigcap_{i\in I} \downarrow_i \bar{s}[i] \neq \emptyset$. Let $\mathbb{S}$ be the set of all compatible states.*

Example 4 The compatible states of $\mathcal{M}$ of Example 2 follow, where we write each function from versions to states $\{(1, s), (2, s'), (3, s'')\}$ as the vector $[s, s', s'']$:

$$\bar{s}_1 = [s_1, s'_3, s''_1] \qquad \bar{s}_2 = [s_1, s'_3, s''_3] \qquad \bar{s}_3 = [s_2, s'_2, s''_2]$$
$$\bar{s}_4 = [s_3, s'_1, s''_4] \qquad \bar{s}_5 = [s_4, s'_3, s''_4]$$

We finally define the succinct representation for the satisfaction set of a CML proposition. More precisely, given ϕ, we compute the set of compatible states $\bar{s}$ that correspond to concrete states satisfying ϕ. The definition is given by recursion, immediately suggesting a model-checking procedure.

Definition 9 (Compatible Satisfaction Set). *Given a* CML *model $\mathcal{M} = \{(\mathcal{K}_i, \downarrow_i)\}_{i\in I}$ with $\mathbb{S}$ its set of compatible states and $\mathcal{K}_i = (S_i, \rightarrow_i, \vdash_i)$ for all $i \in I$, and given $\phi \in \Phi^\bullet$, we let the* compatible satisfaction set *of ϕ with respect to $\mathcal{M}$ be $[\![\phi]\!] \subseteq \mathbb{S}$ recursively defined as follows:*

$$[\![true]\!] = \mathbb{S} \qquad [\![\neg\phi]\!] = \mathbb{S} \setminus [\![\phi]\!] \qquad [\![\phi \wedge \phi']\!] = [\![\phi]\!] \cap [\![\phi']\!]$$
$$[\![\uparrow_i p]\!] = \{\bar{s} \in \mathbb{S} \mid \bar{s}[i] \vdash_i p\} \qquad [\![\uparrow_i \Diamond\phi]\!] = \{\bar{s} \mid \bar{s}' \in [\![\phi]\!] \text{ exists s.t. } \bar{s}[i] \rightarrow_i \bar{s}'[i]\}$$
$$[\![\uparrow_i \blacklozenge\phi]\!] = \{\bar{s} \mid \bar{s}' \in [\![\phi]\!] \text{ exists s.t. } \bar{s}'[i] \rightarrow_i \bar{s}[i]\}$$

Example 5. Consider $\psi_{1,3}$ of Example 3, and the compatible states of Example 4. The compatible satisfaction set for the normal form of $\psi_{1,3}$ is:

$$\begin{aligned}[\![\neg(\uparrow_1 \mathtt{a} \wedge \uparrow_1 \Diamond \uparrow_1 \mathtt{b} \wedge \neg \uparrow_3 \Diamond \uparrow_1 \mathtt{b})]\!] &= \mathbb{S} \setminus (\{\bar{s}_1, \bar{s}_2, \bar{s}_4\} \cap \{\bar{s}_1, \bar{s}_2\} \cap \{\bar{s}_1, \bar{s}_3, \bar{s}_4, \bar{s}_5\}) \\ &= \mathbb{S} \setminus \{\bar{s}_1\} = \{\bar{s}_2, \bar{s}_3, \bar{s}_4, \bar{s}_5\}\end{aligned}$$

Note that $\downarrow_1 \bar{s}_1[1] \cap \downarrow_2 \bar{s}_1[2] \cap \downarrow_3 \bar{s}_3[1] = \{\omega_4\}$, the only counterexample of $\psi_{1,3}$.

The following result guarantees the correctness of the compatible satisfaction set with respect to the satisfaction relation of CML.

Theorem 2. *Given a* CML *model* $\mathcal{M} = \{(\mathcal{K}_i, \downarrow_i)\}_{i \in I}$*, it holds for all* $\omega \in \Omega$ *and* $\phi \in \Phi^\bullet$ *that* $\omega \models_{\mathcal{M}} \phi$ *if and only if* $\omega \in \bigcup_{\bar{s} \in [\![\phi]\!]} \bigcap_{i \in I} \downarrow_i \bar{s}[i]$

3 Representing SEAndroid Policies

In the following, we let Ω be the set of legal file paths of Android devices. As it is clear from the notation, files will be the concrete states for our CML encoding. We also let $\Sigma \supsetneq \Omega$ be the set of all possible entities of an Android system: files, processes, ports. We denote with $\mathbb{L}$ and $\mathbb{L}_{\mathtt{f}} \subsetneq \mathbb{L}$ the sets of all SEAndroid labels and of those associated with files, respectively; $\ell_\perp \in \mathbb{L}_{\mathtt{f}}$ is the distinguished *default* file label used when no other labels can be associated (usually `default_t`). We assume O to be the set of operations controlled by SEAndroid (reading files, searching in directories, etc.). Finally, we let $\mathbb{P}$ be the set of properties appearing in security policies, which we will use as atomic propositions in the CML encoding.

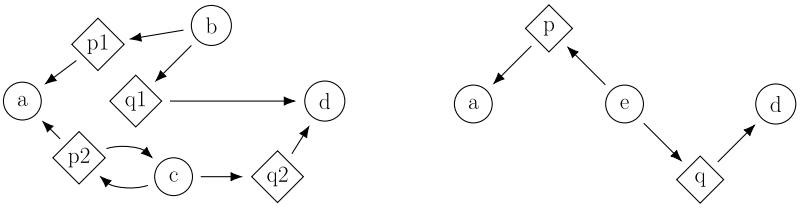
A SEAndroid configuration has two main components: a set of permitted operations between labelled entities and a list of rules for associating files with labels. In addition, we consider a security policy annotating labels with their security properties, such as secrecy or trust level.

Definition 10 (SEAndroid Configuration). *A* SEAndroid configuration $\mathcal{C}$ *is a triple* $(\mathbb{A}, \mathbb{F}, \Gamma)$ *where*

- *the* access control policy $\mathbb{A}$ *is a finite set of triples* $(\ell, o, \ell') \in \mathbb{L} \times O \times \mathbb{L}$*;*
- *the* file labeling policy $\mathbb{F}$ *is a list of pairs* $(\mathcal{L}, \ell)$ *with* $\mathcal{L} \subseteq 2^\Omega$ *a regular language of file paths and* $\ell \in \mathbb{L}_{\mathtt{f}}$ *a file label;*
- *the* security policy $\Gamma \subseteq \mathbb{L}_{\mathtt{f}} \times \mathbb{P}$ *associates labels with predicates in* $\mathbb{P}$*.*

Example 6. Consider the configurations $\mathcal{C}_1 = (\mathbb{A}_1, \mathbb{F}_1, \Gamma_1)$ and $\mathcal{C}_2 = (\mathbb{A}_2, \mathbb{F}_2, \Gamma_2)$ of Fig. 1. We interpret each line `allow b a : file { write };` of the access control policies in Figs. 1c and 1d as a triple $(\mathtt{b}, \mathtt{file}\ \mathtt{write}, \mathtt{a})$ in SELinux notation. The file labelling policies in Figs. 1a and 1b are lists of pairs, where languages are represented as regular expressions. In accordance with Android behaviour, we read the files bottom-up, thus taking a list in reverse order with respect to what is in Fig. 1. The security policies are:

$$\begin{aligned}\Gamma_1 &= \{(a, \mathtt{crit}), (b, \mathtt{usr}), (c, \mathtt{usr}), (d, \mathtt{untr})\} \\ \Gamma_2 &= \{(a, \mathtt{crit}), (e, \mathtt{usr}), (d, \mathtt{untr})\}.\end{aligned}$$



(a) Information flow relation of $\mathbb{A}_1$. (b) Information flow relation of $\mathbb{A}_2$.

Fig. 4. Information flow relation of two configurations: circles and diamonds are file and non-file labels respectively, we omit the arcs obtained by transitivity.

We assume an information flow evaluation function $ife : O \to 2^{\{\triangleright,\triangleleft\}}$ associating each operation o with the directions of the information flows it causes: $\triangleright \in ife(o)$ if o allows the process to send information to the object on which o is performed; $\triangleleft \in ife(o)$ if o allows the process to receive information from it. It is now possible to represent the information flow resulting from a policy.

Definition 11 (Information Flow Relation). *The* information flow relation I *of a SEAndroid access control policy* $\mathbb{A}$ *is the smallest subset of* $\mathbb{L} \times \mathbb{L}$ *such that:*

- *if* $(\ell, o, \ell') \in \mathbb{A}$ *for some* o *with* $\triangleright \in ife(o)$ *then* $(\ell, \ell') \in I$*;*
- *if* $(\ell', o, \ell) \in \mathbb{A}$ *for some* o *with* $\triangleleft \in ife(o)$ *then* $(\ell, \ell') \in I$*;*
- *if* $(\ell, \ell'') \in I$ *and* $(\ell'', \ell') \in I$ *then* $(\ell, \ell') \in I$*.*

Computing the Kripke structure of a SEAndorid configuration amounts to considering the information flow relation over file labels only, and to associate such labels with atomic propositions according to the security policy Γ.

Definition 12 (Kripke Encoding). *The Kripke structure of a SEAndroid configuration* $(\mathbb{A}, \mathbb{F}, \Gamma)$ *with information flow relation* I *is* $(\mathbb{L}_{\mathtt{f}}, I \cap (\mathbb{L}_{\mathtt{f}} \times \mathbb{L}_{\mathtt{f}}), \Gamma)$.

Example 7. The graphs above in Fig. 2 represent the Kripke structures of the two configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ of Fig. 1. The transitions are obtained as the subset over file labels of the information flow relations depicted in Fig. 4; atomic propositions are associated according to the security policies Γ_1 and Γ_2.

Given $\mathbb{F}$, the file labelling policy of the SEAndroid configuration, we define $\lambda_{\mathbb{F}}$ to be the function used by the operating system to associate labels with files. Intuitively, given a file with path ω, the rules of $\mathbb{F}$ are processed sequentially (i.e. in reverse order with respect to the file context). For each pair $(\mathcal{L}, \ell)$, the regular expression representing $\mathcal{L}$ is considered: if $\omega \in \mathcal{L}$, the label ℓ is associated with ω, otherwise the process moves to the next pair.

Definition 13 (File Labeling Function). *Given a file labeling policy* $\mathbb{F}$*, its* file labeling function $\lambda_{\mathbb{F}} \colon \Omega \longrightarrow \mathbb{L}_{\mathtt{f}}$ *is defined as* $\lambda_{\mathbb{F}}(\omega) = \ell$ *if* $(\mathcal{L}, \ell)$ *is the first pair in* $\mathbb{F}$ *such that* $\omega \in \mathcal{L}$*, and* $\lambda_{\mathbb{F}}(\omega) = \ell_\perp$ *if no* $\mathcal{L}$ *of* $\mathbb{F}$ *contains* ω*.*

Example 8. The file labeling functions of $\mathcal{C}_1$ and $\mathcal{C}_2$ in Fig. 1 follow, where $\mathcal{L}_a$ and $\mathcal{L}_e$ are the languages recognized by `(A/.*)|(.*/b)` and `.*/a`, respectively.

$$\lambda_{\mathbb{F}_1}(\omega) = \begin{cases} b & \text{if } \omega = \texttt{C/a} \\ c & \text{if } \omega = \texttt{B/b} \\ d & \text{if } \omega = \texttt{C/b} \\ a & \text{if } \omega \in \mathcal{L}_a \setminus \{\texttt{C/a}, \texttt{B/b}, \texttt{C/b}\} \\ \ell_\perp & \text{otherwise} \end{cases} \qquad \lambda_{\mathbb{F}_2}(\omega) = \begin{cases} d & \text{if } \omega = \texttt{C/b} \\ e & \text{if } \omega \in \mathcal{L}_e \setminus \{\texttt{C/b}\} \\ a & \text{if } \omega \in \mathcal{L}_a \setminus (\mathcal{L}_e \cup \{\texttt{C/b}\}) \\ \ell_\perp & \text{otherwise} \end{cases}$$

The graphs below in Fig. 2 are pictorial representations of the concretisations of the Kripke structures of the two configurations above (see Example 7).

Finally, the CML model of a pair of SEAndroid configurations can be computed by associating their Kripke structures with the label evaluation obtained by inverting the file labelling function.

Definition 14 (CML **Encoding).** *The* CML *model of two SEAndroid configurations* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *is* $\{(\mathcal{K}_1, \downarrow_1), (\mathcal{K}_2, \downarrow_2)\}$*, where for* $i = 1, 2$*:* $\mathcal{K}_i$ *is the Kripke structure of* $\mathcal{C}_i$*;* $\downarrow_i : \mathbb{L}_\texttt{f} \longrightarrow 2^\Omega$ *is such that* $\downarrow_i(\ell) = \{\omega \mid \lambda_{\mathbb{F}_i}(\omega) = \ell\}$ *for all* $\ell \in \mathbb{L}_\texttt{f}$*, with* $\mathbb{F}_i$ *the file labeling policy of* $\mathcal{C}_i$*.*

We now use this encoding to compare two instances of SEAndroid in terms of their information flow through the procedure of Sect. 2.

Example 9. Consider the CML model for $\mathcal{C}_1$ and $\mathcal{C}_2$ of Fig. 1. We start by computing the set of compatible states $\mathbb{S}$. To simplify our notation, we represent compatible states as pairs $[\ell, \ell']$, with ℓ associated with the first configuration and ℓ' with the second. A pair of file labels $[\ell, \ell']$ is included in $\mathbb{S}$ if there exists at least a filepath that is associated with ℓ in the first policy and with ℓ' in the second one. For example, $[a, e] \in \mathbb{S}$ because of `A/a`, while $[c, e] \notin \mathbb{S}$ because c labels only `B/b` in $\mathbb{F}_1$ and e labels only filepaths ending with `a` in $\mathbb{F}_2$. Simple computations suffice for showing that $\mathbb{S} = \{[a, a], [a, e], [b, e], [c, a], [d, d]\}$.

Consider the CML formula $\phi = \uparrow_1(\Diamond\texttt{untr}) \Rightarrow \uparrow_2(\texttt{usr} \wedge \Diamond\uparrow_1\texttt{untr})$ of Sect. 1. To check the validity of ϕ in our model, we take $\neg\phi$ and normalise it (Theorem 1). The result is $\psi = \uparrow_1\Diamond\uparrow_1\texttt{untr} \wedge (\neg\uparrow_2\texttt{usr} \vee \neg\uparrow_2\Diamond\uparrow_1\texttt{untr})$. We compute the compatible satisfaction set of ψ according to Definition 9:

$$\begin{aligned} [\![\psi]\!] &= [\![\uparrow_1\Diamond\uparrow_1\texttt{untr}]\!] \setminus ([\![\uparrow_2\texttt{usr}]\!] \cup [\![\uparrow_2\Diamond\uparrow_1\texttt{untr}]\!]) \\ &= \{[b, e], [c, a]\} \setminus (\{[a, e], [b, e]\} \cup \{[a, e], [b, e]\}) = \{[c, a]\}. \end{aligned}$$

The property is not satisfied, and concrete counterexamples are those files labelled as c in $\mathcal{C}_1$ and as a in $\mathcal{C}_2$, i.e. $\downarrow_1(c) \cap \downarrow_2(a) = \{\texttt{B/b}\}$.

4 Implementation and Evaluation of *Mordente*

We implement our verification framework in the tool *Mordente*, available at [1]. Below, we discuss its internals and evaluate it on real-world policies.

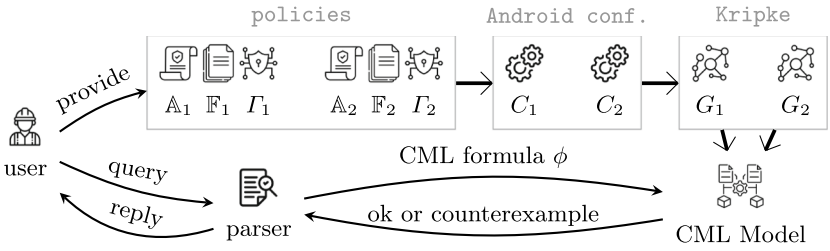


Fig. 5. Workflow of *Mordente*.

Algorithm 1. Compute the state evaluation $\downarrow_i$ of the i-th configuration.

Input: An ordered list $\mathbb{F}$ of pairs (ℓ, α) with ℓ a string and α an automaton
Output: A dictionary $\mathbb{D}$ from strings to automata

1: $\zeta \leftarrow \mathcal{L}(\epsilon)$ ▷ ζ is the language of previously seen file paths
2: $\mathbb{D}[\ell] \leftarrow \emptyset$ **for all** ℓ
3: **for all** pairs (ℓ, α) in $\mathbb{F}$ **do**
4: $\quad \beta \leftarrow \alpha \cap \zeta^c$ ▷ ζ^c is the complement of ζ
5: $\quad \mathbb{D}[\ell] \leftarrow \mathbb{D}[\ell] \cup \beta$
6: $\quad \zeta \leftarrow \zeta \cup \beta$
7: **return** $\mathbb{D}$

***Mordente* Internals.** The workflow of *Mordente* is in Fig. 5. The user provides two compiled SEAndroid policies to be compared, each accompanied by its file context specification and its security policy Γ. For each policy, *Mordente* constructs a *SEAndroid configuration* $\mathcal{C} = (\mathbb{A}, \mathbb{F}, \Gamma)$ and then its *Kripke structure* $\mathcal{K}$. It then computes the *SEAndroid* CML *Model* for the pair of configurations, and waits for the user to submit CML queries for the differential verification.

More in detail, *Mordente* processes its input as follows. To build the access control policy $\mathbb{A}$, we extract all the *allow* rules from the compiled policy using setools [3]. Constructing the file labelling policy $\mathbb{F}$ requires additional work. We first process the file context entries in reverse order, as the Android system does. As a result, we obtain a list of pairs $(\texttt{regex}, \ell)$, where $\ell \in \mathbb{L}_{\texttt{f}}$ is a label and `regex` a regular expression of file paths. We then translate each `regex` into an equivalent finite state automaton using *libmata* [6]. To compute $\downarrow_i(\ell)$, the regular language of file paths associated with ℓ, we apply Algorithm 1, which iteratively refines each automaton to remove paths already matched by earlier entries, resolving shadowing between overlapping file context regexes. More precisely, Algorithm 1 returns a dictionary $\mathbb{D}$ from labels to automata, such that the language of $\mathbb{D}[\ell]$ is disjoint from the one of $\mathbb{D}[\ell']$ if $\ell \neq \ell'$. Note that if the same label ℓ appears more than once in the file context, we take the union of the resulting automata.

For efficiency, *Mordente* does not explicitly construct the full information flow relation I of Definition11. Instead, it builds a graph G where nodes are SEAndroid labels and edges are a compact representation of I that omits transitive edges. More precisely, for each triple $(s, o, s') \in \mathbb{A}$, we add to G the edge (s, s')

Algorithm 2. Removes non-file labels from G while preserving connectivity.

Input: A graph $G = (N, E)$
Output: G where all the nodes are file labels

```
for all n ∈ N such that n is not a file label do
    for all (u, n) ∈ E with u ≠ n do
        for all (n, v) ∈ E with n ≠ v do                ▷ u and v are connected
            if not (u, v) ∈ E then E ← E ∪ {(u, v)}
    Remove n from G
return G
```

Table 1. *Mordente* experimental evaluation.

(a) Generation of the Kripke structures.

	file context lines	allow rules	build time (s)
① Pixel Comet (14)	1327	41296	436.24
② Pixel Comet (15)	1351	42744	454.74
③ Pixel Comet (16)	1397	45104	494.26
④ Xiaomi 11T	1940	52122	504.37
⑤ Xiaomi 14T	2264	63673	1188.80
⑥ Xiaomi 15T	2271	56784	1141.90

(b) Differential analysis.

	compatible states	build time (s)	query time (s)
① ↔ ②	769	17.52	2.18
② ↔ ③	814	17.51	2.39
① ↔ ④	1568	37.46	16.28
② ↔ ⑤	1592	38.11	13.96
② ↔ ⑥	1608	49.80	21.97

if $\triangleright \in \mathit{ife}(o)$ and (s', s) if $\triangleleft \in \mathit{ife}(o)$. We remove from G the non-file labels, i.e. those not in $\mathbb{D}$, while updating the edges to preserve connectivity: if two file labels were connected via a path possibly traversing non-file labels, the pruned graph maintains a path between them (see Algorithm 2). Formally, after this pruning of G, its edges compactly represent the relation $I \cap (\mathbb{L}_{\mathtt{f}} \times \mathbb{L}_{\mathtt{f}})$. Then, the Kripke structure $\mathcal{K}$ is obtained by considering the security policy Γ and associating atomic predicates to nodes of G accordingly.

After constructing both Kripke structures, *Mordente* computes the set of compatible states of the CML Model: these are pairs of labels (ℓ_1, ℓ_2) such that the intersection of their corresponding automata recognises a non-empty language. Finally, once the user submits a CML formula ϕ, the verification proceeds as follows. We take $\neg\phi$ and compute its normal form $\psi^\bullet$ (Theorem 1). A custom model-checking procedure, based on Definition 9, computes the set of compatible states $[\![\psi^\bullet]\!]$ that falsify ϕ (note in passing that the modalities $\uparrow_i \Diamond$ and $\uparrow_i \blacklozenge$ are interpreted as reachability over the graph G). If $[\![\psi^\bullet]\!]$ is empty, the property ϕ holds; otherwise, a counterexample is generated by instantiating the offending states in $[\![\psi^\bullet]\!]$ with concrete file paths.

Experimental Evaluation. We experimentally evaluated *Mordente* on a collection of policies from different vendors and Android versions. We automated the tests with a script, available at [1], which takes in input the Android firmware images of the systems to compare, it extracts the SEAndroid policy configuration

$$\phi_1 = \uparrow_j \texttt{critical} \Rightarrow \uparrow_i \texttt{critical}$$
$$\phi_2 = \uparrow_j \Diamond \texttt{critical} \Rightarrow \uparrow_i \Diamond \uparrow_j \texttt{critical}$$
$$\phi_3 = \uparrow_j (\texttt{untrusted} \wedge \Diamond \texttt{critical}) \Rightarrow \uparrow_i \Diamond \uparrow_j \texttt{critical}$$
$$\phi_4 = \uparrow_j (\texttt{untrusted} \wedge \Diamond \texttt{critical}) \Rightarrow \uparrow_i (\texttt{untrusted} \wedge \Diamond \texttt{critical})$$
$$\phi_5 = \uparrow_j (\texttt{untrusted} \wedge \Diamond \texttt{critical}) \wedge \uparrow_i \texttt{untrusted} \Rightarrow \uparrow_i \Diamond \texttt{critical}$$
$$\phi_6 = \uparrow_j (\texttt{critical} \wedge \blacklozenge \texttt{untrusted}) \Rightarrow \uparrow_i (\uparrow_j \texttt{critical} \wedge \blacklozenge \uparrow_j \texttt{untrusted})$$
$$\phi_7 = (\uparrow_j \Diamond \texttt{critical} \wedge \uparrow_i \texttt{untrusted}) \Rightarrow \uparrow_j \texttt{secure}$$

Fig. 6. CML formulas used for differential analysis of Sect. 4.

files (precompiled policy, file contexts, and information about the Android version). The user can choose to perform either a *vertical* or an *horizontal* analysis. In the first case, policies are from the same vendor but from different Android versions, and are compared incrementally, each version with the next one; in the second case, the policies are from different vendors (presumably of the same Android version) and are compared in every possible combination.

During the analysis, *Mordente* is first invoked once for each SEAndroid configuration to compute its Kripke structure, which is then saved for future use. Whereas the policies for access control and file labelling are extracted from the Android images, the security policy Γ of each version is defined according to the following heuristic: we considered the predicates *untrusted*, *secure*, and *critical*, and associated them with file labels that contain the predicate as a substring. In addition, we manually annotate well-known Android files with the most appropriate security properties. Then, the tool is applied to each pair of policies to compute their CML Model and compatible states, and finally to check the validity of some submitted queries.

We evaluated *Mordente* on 6 configurations as reported in Table 1: three are from major Android releases (14, 15, 16) of the firmware for a Pixel 9 Pro Fold (*Comet*) [8]; three others are from different Xiaomi devices (11T, 14T, 15T) [17], selected with a similar Android version (14, 15). These configurations were selected because they are released by two major OEMs: the Pixel devices are produced by Google, which leads Android development; while the Xiaomi devices are from one of the most widely used Android vendors worldwide. For any analyzed pair of versions i and j of SEAndroid configurations, we verified the seven CML formulas in Fig. 6, encoding the following intuitive properties: ϕ_1 says that critical files should not increase, encoding the desired minimality property of the trusted computing base of operating systems; ϕ_2 says that files influencing critical resources do not increase, as they can possibly lead to integrity violations; ϕ_3 checks that untrusted files influencing critical entities do not increase, thus weakening the previous formula; ϕ_4 and ϕ_5 are variations of the previous one discussed in Example 2; ϕ_6 checks that critical files influenced by some untrusted entity do not increase, limiting the impact of possibly malicious resources; ϕ_7 verifies that files that were once untrusted and can now influence critical entities

must be labelled as secure, i.e. it allows declassification only when explicitly documented.

The experiments were conducted on a virtual machine with 4 cores of an Intel Core i7-10700 K processor (3.80 GHz) and 16GB of RAM, running Ubuntu 24.04. In more detail, Table 1a shows the size of each considered configuration (the number of file context lines and *allow* rules) as well as the time it takes on average over 10 executions to construct its Kripke structure. We analysed Pixel configurations vertically (comparing each version with the next one); whereas each Xiaomi configuration was assessed against the same Android major version of Pixel. Table 1b reports the results of our differential analysis over the selected pairs of configurations: the number of compatible states, the time to compute the resulting CML model, and the time to verify 7 formulas discussed above (average over 10 executions).

Overall, the time needed to compute the Kripke structure is acceptable for both vendors. Furthermore, the time required to compute a CML model is considerably smaller, and the queries are almost instant. In addition, *Mordente* is capable of storing the computed Kripke structure and reload it when needed, vastly reducing the expected loading times by a factor of 20 on average.

5 Related Work

The security of SEAndroid and SELinux has been extensively studied in the literature from various perspectives. Below, we first discuss proposals that examine how SEAndroid policies have evolved over the years and across various vendors. Then, we consider research on verifying information-flow properties in SELinux.

Im et al. [11] perform a historical analysis on the AOSP repository to understand the evolution of SEAndroid policy. In particular, the authors propose a metric for the complexity of policies and measure different commits of the main branch of the repository: the complexity of SEAndroid policies has been growing exponentially over time, confirming the necessity of specialised tools and techniques for their analysis. Yu et al. [18] investigate the status of SEAndroid policy customisation, namely, the differences in terms of permissions between the official AOSP policy and the versions of vendors. They propose the tool SEPAL to automatically predict whether the customised policy rules are potentially risky or unnecessary, using natural language processing techniques. Posemato et al. [14] perform a longitudinal study on Android OEM customisations with the goal of determining if these customisations lead to compatibility and security problems. They consider the various requirements imposed by Google to brand a device "Android", including some over SEAndroid. They highlighted that many customisations removed some `neverallow` rules from the AOSP policy, weakening the security of the resulting policy. The papers above compare SEAndroid policies using ad hoc algorithms that analyse individual rules and their granted permissions. In contrast, our work assigns a formal semantics to SEAndroid policies and verifies them not at the level of single rules, but in terms of the information flows induced by the permissions collectively granted across

multiple rules. Moreover, our logic and model-checking procedure enable policy analysts to express and verify a variety of relational queries over pairs of policies, offering greater expressiveness than rule-by-rule comparison.

Some tools have been proposed to compare two SELinux policies. The `sediff` tool is a command-line utility, part of the SETools (SELinux Policy Analysis Tools) suite, used to compare and find the semantic differences between two SELinux policies. The tool operates at the entity level and highlights differences in various policy elements, including rules, types, and attributes. More precisely, given two policies, the tool can detect if an element exists only in one of the two policies or if it has been modified, namely, it exists in both policies, but the associated permissions have changed. The V3SPA [9] is a visualisation tool for SELinux policies that supports policy analysts in understanding the meaning of a policy and performing differential policy analysis, highlighting the changes between two policy versions. Policies are depicted as graphs, where nodes represent subjects and objects, and edges represent the permissions associated with them. These graphs can be manually inspected by analysts. These tools offer limited analysis capabilities because they either focus on individual permissions or require policy analysts to manually compare graph representations of the policies. In contrast, our approach supports the specification of more expressive information-flow properties and automatically checks them, thereby providing better scalability and generality. In addition, our work is the first to compare permissions in terms of concrete files, considering possibly different labelling policies, whereas previous approaches only reason at the level of labels.

Several papers target information flow in SELinux policies. Radika et al. [15] analyse SELinux configurations to identify potentially dangerous information flows, defined by the authors as those from some entity a to some b such that a `neverallow` rule prohibits a direct read access from b to a. They developed a tool to statically investigate such information flows, and applied it to the AOSP policy; Jaeger et al. [12] analyse the SELinux example policy for Linux 2.4.19, identifying the trusted computing base and studying its integrity. Hernandez et al. [10] investigate the Android policy environment, including SEAndroid, UNIX-style DAC, and Linux capabilities, and propose BigMAC, a framework that unifies all policy layers to produce a large, fine-grained attack graph. A logic-based query engine is then used to prune infeasible paths and unused types. Similarly, Lee et al. [13] describe PolyScope, a tool for analysing Android systems and identifying resources that attackers can modify, with and without policy manipulations. For each integrity violation detected, PolyScope characterises how attacks may be mounted. Ceragioli et al. [5] enriched the intermediate language CIL, used to specify SELinux configurations with a formal semantics and mechanisms to allow users to specify and verify information-flow requirements. All these papers analyse one policy at a time and therefore cannot detect differences in permitted information flow across different policies. In contrast, our proposal enables the specification and verification of information-flow properties over pairs of policies, thus supporting change-impact analysis of SEAndroid configurations.

6 Conclusion

We have presented a differential verification framework for SEAndroid that enables reasoning across multiple versions or vendor variants of a policy, capturing how changes in the permissions and labelling affect the information-flow properties. Our framework is built on CML, a modal logic that we designed explicitly to express comparative properties across pairs of Kripke structures. We have formalised SEAndroid policies as Kripke structures and have defined a suitable CML model-checking algorithm for differential reasoning on them. We have implemented our approach in the *Mordente* tool and evaluated its scalability on real-world SEAndroid policies from different vendors and Android releases. Our experiments show that differential verification is feasible in practice. To the best of our knowledge, *Mordente* is the first tool to compare SEAndroid configurations in a formally grounded way that simultaneously accounts for changes in both the permission and labelling policies. A key advantage of *Mordente* is that it allows proving the preservation of undocumented security properties.

Future work includes using *Mordente* to perform a large-scale security analysis of SEAndroid customisation to identify security-relevant deviations. Moreover, we plan to extend *Mordente* by integrating automated repair strategies for policy regressions.

Acknowledgements. Work supported by the project SERICS (PE00000014, PNRR M4C2 I1.3, CUP: D67G22000340001) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

1. Mordente: tool for differential verification of information flow in SEAndroid policies. https://doi.org/10.5281/zenodo.18606935. Repository: https://github.com/edoxthebest/Mordente
2. Selinux project. https://selinuxproject.org
3. Setools: policy analysis tools for selinux. https://github.com/SELinuxProject/setools
4. The SELinux Notebook - Kernel Policy Language. https://github.com/SELinuxProject/selinux-notebook/blob/main/src/kernel_policy_language.md#kernel-policy-language
5. Ceragioli, L., Galletta, L., Degano, P., Basin, D.: Specifying and verifying information flow control in selinux configurations. ACM Trans. Priv. Secur **27**(4), (2024)
6. Chocholatý, D., et al.: Mata: a fast and simple finite automata library. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 130–151. Springer Nature Switzerland, Cham (2024)
7. Drake, J.: Stagefright: scary code in the heart of android (2015). https://blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf. BlackHat USA

8. Google: full Ota images for nexus and pixel devices. https://developers.google.com/android/ota
9. Gove, R.: V3spa: a visual analysis, exploration, and diffing tool for selinux and seandroid security policies. In: 2016 IEEE Symposium on Visualization for Cyber Security (VizSec), pp. 1–8. (2016). https://doi.org/10.1109/VIZSEC.2016.7739580
10. Hernandez, G., Tian, D.J., Yadav, A.S., Williams, B.J., Butler, K.R.: BigMAC: fine-grained policy analysis of android firmware. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 271–287. USENIX Association (2020). https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez
11. Im, B., Chen, A., Wallach, D.S.: An historical analysis of the seandroid policy evolution. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 629–640. ACSAC '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3274694.3274709
12. Jaeger, T., Sailer, R., Zhang, X.: Analyzing integrity protection in the selinux example policy. In: Procs 12th USENIX Security Symposium, USENIX Association, Washington, D.C., USA (2003)
13. Lee, Y.T., et al.: PolyScope: multi-policy access control analysis to compute authorized attack operations in Android systems. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 2579–2596. USENIX Association (2021). https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yu-tsung
14. Possemato, A., Aonzo, S., Balzarotti, D., Fratantonio, Y.: Trust, but verify: a longitudinal analysis of android OEM compliance and customization. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 87–102 (2021). https://doi.org/10.1109/SP40001.2021.00074
15. Radhika, B.S., Kumar, N.V.N., Shyamasundar, R.K., Vyas, P.: Consistency analysis and flow secure enforcement of selinux policies. Comput. Secur. **94**, 101816 (2020)
16. Seri, B., Vishnepolsky, G.: The dangers of bluetooth implementations: unveiling zero day vulnerabilities and security flaws in modern bluetooth stacks. Technical Report, Armis INC. White Paper (2023). https://info.armis.com/rs/645-PDC-047/images/BlueBorne
17. Xiaomi: Xiaomi community. https://c.mi.com/global/miuidownload/index
18. Yu, D., et al.: Sepal: towards a large-scale analysis of seandroid policy customization. In: Proceedings of the Web Conference 2021, pp. 2733–2744. WWW '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3442381.3450007